11 – The Environment

As we discussed earlier, the shell maintains a body of information during our shell session called the *environment*. Programs use data stored in the environment to determine facts about the system's configuration. While most programs use *configuration files* to store program settings, some programs also look for values stored in the environment to adjust their behavior. Knowing this, we can use the environment to customize our shell experience.

In this chapter, we will work with the following commands:

- `printenv` – Print part or all of the environment
- `set` – Set shell options
- `export` – Export environment to subsequently executed programs
- `alias` – Create an alias for a command

What is Stored in the Environment?

The shell stores two basic types of data in the environment; though, with `bash,` the types are largely indistinguishable. They are *environment variables* and *shell variables*. Shell variables are bits of data placed there by `bash`, and environment variables are everything else. In addition to variables, the shell stores some programmatic data, namely *aliases* and *shell functions*. We covered aliases in Chapter 5, “Working with Commands.” and we will cover shell functions (which are related to shell scripting) in Part 4.

Examining The Environment

To see what is stored in the environment, we can use either the `set` builtin in `bash` or the `printenv` program. The `set` command will show both the shell and environment variables, while `printenv` will only display the latter. Since the list of environment contents will be fairly long, it is best to pipe the output of either command into `less.`

```
[me@linuxbox ~]$ printenv | less
```

Doing so, we should get that looks like this:

```
USER=me
PAGER=less
LSCOLORS=Gxfxcxdxbxegedabagacad
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/usr/local/games
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
JOB=dbus
PWD=/home/me
XMODIFIERS=@im=ibus
GNOME_KEYRING_PID=1850
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
 MASTER_HOST=linuxbox
IM_CONFIG_PHASE=1
COMPIZ_CONFIG_PROFILE=ubuntu
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
XDG_SEAT=seat0
HOME=/home/me
SHLVL=2
LANGUAGE=en_US
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LESS=-R
LOGNAME=me
COMPIZ_BIN_PATH=/usr/bin/
LC_CTYPE=en_US.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/
usr/share/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-IwaesmWaT0
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
```

What we see is a list of environment variables and their values. For example, we see a variable called `USER`, which contains the value `me`. The `printenv` command can also list the value of a specific variable.

```
[me@linuxbox ~]$ printenv USER
me
```

The `set` command, when used without options or arguments, will display both the shell and environment variables, as well as any defined shell functions. Unlike `printenv`, its output is courteously sorted in alphabetical order.

```
[me@linuxbox ~]$ set | less
```

It is also possible to view the contents of a variable using the `echo` command, like this:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

One element of the environment that neither `set` nor `printenv` displays is aliases. To see them, enter the `alias` command without arguments.

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-
dot --show-tilde'
```

Some Interesting Variables

The environment contains quite a few variables, and though the environment will differ from the one presented here, we will likely see the variables listed in Table 11-1 in our environment.

Table 11-1: Environment Variables

Variable	Contents
`DISPLAY`	The name of the display if we are running a graphical environment. Usually this is ":0", meaning the first display generated by the X server.

`EDITOR`	The name of the program to be used for text editing.
`SHELL`	The name of the user's default shell program.
`HOME`	The pathname of your home directory.
`LANG`	Defines the character set and collation order of your language.
`OLDPWD`	The previous working directory.
`PAGER`	The name of the program to be used for paging output. This is often set to `/usr/bin/less`.
`PATH`	A colon-separated list of directories that are searched when we enter the name of a executable program.
`PS1`	This stands for "prompt string 1." This defines the contents of the shell prompt. As we will later see, this can be extensively customized.
`PWD`	The current working directory.
`TERM`	The name of your terminal type. Unix-like systems support many terminal protocols; this variable sets the protocol to be used with your terminal emulator.
`TZ`	Specifies your time zone. Most Unix-like systems maintain the computer's internal clock in *Coordinated Universal Time* (UTC) and then display the local time by applying an offset specified by this variable.
`USER`	Your username.

Don't worry if some of these values are missing. They vary by distribution.

How Is The Environment Established?

When we log on to the system, the `bash` program starts, and reads a series of configuration scripts called *startup files*, which define the default environment shared by all users. This is followed by more startup files in our home directory that define our personal environment. The exact sequence depends on the type of shell session being started. There are two kinds.

- **A login shell session** A login shell session is one in which we are prompted for our username and password. This happens when we start a virtual console session, for example.
- **A non-login shell session** A non-login shell session typically occurs when we

launch a terminal session in the GUI.

Login shells read one or more startup files as shown in Table 11-2.

Table 11-2: Startup Files for Login Shell Sessions

File	Contents
/etc/profile	A global configuration script that applies to all users.
~/.bash_profile	A user's personal startup file. This can be used to extend or override settings in the global configuration script.
~/.bash_login	If ~/.bash_profile is not found, bash attempts to read this script.
~/.profile	If neither ~/.bash_profile nor ~/.bash_login is found, bash attempts to read this file. This is the default in Debian-based distributions, such as Ubuntu.

Non-login shell sessions read the startup files listed in Table 11-3.

Table 11-3: Startup Files for Non-Login Shell Sessions

File	Contents
/etc/bash.bashrc	A global configuration script that applies to all users.
~/.bashrc	A user's personal startup file. It can be used to extend or override settings in the global configuration script.

In addition to reading the startup files in Table 11-3, non-login shells inherit the environment from their parent process, usually a login shell.

Take a look and see which of these startup files are installed. Remember—since most of the filenames listed above start with a period (meaning that they are hidden), we will need to use the “-a” option when using `ls`.

The `~/.bashrc` file is probably the most important startup file from the ordinary user’s point of view, since it is almost always read. Non-login shells read it by default and most startup files for login shells are written in such a way as to read the `~/.bashrc` file as well.

What's in a Startup File?

If we take a look inside a typical `.bash_profile` (taken from a CentOS 6 system), it looks something like this:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
export PATH
```

Lines that begin with a "#" are *comments* and are not read by the shell. These are there for human readability. The first interesting thing occurs on the fourth line, with the following code:

```
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

This is called an *if compound command*, which we will cover fully when we get to shell scripting in Part 4, but for now, here is a translation:

```
If the file "~/.bashrc" exists, then
   read the "~/.bashrc" file.
```

We can see that this bit of code is how a login shell gets the contents of `.bashrc`. The next thing in our startup file has to do with the `PATH` variable.

Ever wonder how the shell knows where to find commands when we enter them on the command line? For example, when we enter `ls`, the shell does not search the entire computer to find `/bin/ls` (the full pathname of the `ls` command); rather, it searches a list of directories that are contained in the `PATH` variable.

The `PATH` variable is often (but not always, depending on the distribution) set by the `/etc/profile` startup file with this code:

```
PATH=$PATH:$HOME/bin
```

PATH is modified to add the directory $HOME/bin to the end of the list. This is an example of parameter expansion, which we touched on in Chapter 7. "Seeing the World As tthe Shell Sees It." To demonstrate how this works, try the following:

```
[me@linuxbox ~]$ foo="This is some "
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo"text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Using this technique, we can append text to the end of a variable's contents.

By adding the string $HOME/bin to the end of the PATH variable's contents, the directory $HOME/bin is added to the list of directories searched when a command is entered. This means that when we want to create a directory within our home directory for storing our own private programs, the shell is ready to accommodate us. All we have to do is call it bin, and we're ready to go.

> **Note**: Many distributions provide this PATH setting by default. Debian based distributions, such as Ubuntu, test for the existence of the ~/bin directory at login and dynamically add it to the PATH variable if the directory is found.

Lastly, we have:

```
export PATH
```

The export command tells the shell to make the contents of PATH available to child processes of this shell.

Modifying the Environment

Since we know where the startup files are and what they contain, we can modify them to customize our environment.

Which Files Should We Modify?

As a general rule, to add directories to your `PATH` or define additional environment variables, place those changes in `.bash_profile` (or the equivalent, according to your distribution; for example, Ubuntu uses `.profile`). For everything else, place the changes in `.bashrc`.

> **Note:** Unless you are the system administrator and need to change the defaults for all users of the system, restrict your modifications to the files in your home directory. It is certainly possible to change the files in `/etc` such as `profile`, and in many cases it would be sensible to do so, but for now, let's play it safe.

Text Editors

To edit (i.e., modify) the shell's startup files, as well as most of the other configuration files on the system, we use a program called a *text editor*. A text editor is a program that is, in some ways, like a word processor in that it allows us to edit the words on the screen with a moving cursor. It differs from a word processor by only supporting pure text and often contains features designed for writing programs. Text editors are the central tool used by software developers to write code and by system administrators to manage the configuration files that control the system.

A lot of different text editors are available for Linux; most systems have several installed. Why so many different ones? Because programmers like writing them and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories: graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called `gedit`, which is usually called “Text Editor” in the GNOME menu. KDE usually ships with three, which are (in order of increasing complexity) `kedit`, `kwrite`, and `kate`.

There are many text-based editors. The popular ones we'll encounter are `nano`, `vi`, and `emacs`. The `nano` editor is a simple, easy-to-use editor designed as a replacement for the `pico` editor supplied with the PINE email suite. The `vi` editor (which on most Linux systems replaced by a program named `vim`, which is short for “vi improved”) is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The `emacs` editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While readily available, it is seldom installed on most Linux systems by default.

Using a Text Editor

Text editors can be invoked from the command line by typing the name of the editor followed by the name of the file we want to edit. If the file does not already exist, the editor will assume that we want to create a new file. Here is an example using `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

This command will start the `gedit` text editor and load the file named "some_file", if it exists.

Graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text-based text editor, `nano`. Let's fire up `nano` and edit the `.bashrc` file. But before we do that, let's practice some "safe computing." Whenever we edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess up the file while editing. To create a backup of the `.bashrc` file, do this:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

It doesn't matter what we call the backup file; just pick an understandable name. The extensions ".bak", ".sav", ".old", and ".orig" are all popular ways of indicating a backup file. Oh, and remember that `cp` will *overwrite existing files* silently.

Now that we have a backup file, we'll start the editor.

```
[me@linuxbox ~]$ nano .bashrc
```

Once `nano` starts, we'll get a screen like this:

```
  GNU nano 2.0.3           File: .bashrc

# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
```

```
# User specific aliases and functions









                              [ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

Note: If your system does not have `nano` installed, you may use a graphical editor instead.

The screen consists of a header at the top, the text of the file being edited in the middle, and a menu of commands at the bottom. Since `nano` was designed to replace the text editor supplied with an email client, it is rather short on editing features.

The first command we should learn in any text editor is how to exit the program. In the case of `nano`, we press `Ctrl-x` to exit. This is indicated in the menu at the bottom of the screen. The notation `^X` means `Ctrl-x`. This is a common notation for control characters used by many programs.

The second command we need to know is how to save our work. With `nano` it's `Ctrl-o`. With this knowledge, we're ready to do some editing. Using the down arrow key and/or the `PageDown` key, move the cursor to the end of the file, and then add the following lines to the `.bashrc` file:

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Note: Your distribution may already include some of these, but duplicates won't hurt anything.

Table 11-4 details the meaning of our additions:

Table 11-4: Additions to Our `.bashrc`

Line	Meaning
`umask 0002`	Sets the `umask` to solve the problem with the shared directories we discussed in Chapter 9, "Permissions."
`export HISTCONTROL=ignoredups`	Causes the shell's history recording feature to ignore a command if the same command was just recorded.
`export HISTSIZE=1000`	Increases the size of the command history from the usual default of 500 lines to 1,000 lines.
`alias l.='ls -d .* --color=auto'`	Creates a new command called `l.`, which displays all directory entries that begin with a dot.
`alias ll='ls -l --color=auto'`	Creates a new command called `ll`, which displays a long-format directory listing.

As we can see, many of our additions are not intuitively obvious, so it would be a good idea to add some comments to our `.bashrc` file to help explain things to the humans. Using the editor, change our additions to look like this:

```
# Change umask to make directory sharing easier
umask 0002

# Ignore duplicates in command history and increase
# history size to 1000 lines
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Add some helpful aliases
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Ah, much better! With our changes complete, press `Ctrl-o` to save our modified `.bashrc` file, and press `Ctrl-x` to exit `nano`.

Why Comments Are Important

Whenever you modify configuration files it's a good idea to add some comments to document your changes. Sure, you'll probably remember what you changed tomorrow, but what about six months from now? Do yourself a favor and add some comments. While you're at it, it’s not a bad idea to keep a log of what changes you make.

Shell scripts and `bash` startup files use a “#” symbol to begin a comment. Other configuration files may use other symbols. Most configuration files will have comments. Use them as a guide.

You will often see lines in configuration files that are *commented out* to prevent them from being used by the affected program. This is done to give the reader suggestions for possible configuration choices or examples of correct configuration syntax. For example, the `.bashrc` file of Ubuntu 18.04 contains these lines:

```
# some more ls aliases
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

The last three lines are valid alias definitions that have been commented out. If you remove the leading “#” symbols from these three lines, a technique called *uncommenting*, you will activate the aliases. Conversely, if you add a “#” symbol to the beginning of a line, you can deactivate a configuration line while preserving the information it contains.

Activating Our Changes

The changes we have made to our `.bashrc` will not take effect until we close our terminal session and start a new one because the `.bashrc` file is only read at the beginning of a session. However, we can force `bash` to reread the modified `.bashrc` file with the following command:

```
[me@linuxbox ~]$ source ~/.bashrc
```

After doing this, we should be able to see the effect of our changes. Try one of the new aliases.

```
[me@linuxbox ~]$ ll
```

Summing Up

In this chapter, we learned an essential skill—editing configuration files with a text editor. Moving forward, as we read man pages for commands, take note of the environment variables that commands support. There may be a gem or two. In later chapters, we will learn about shell functions, a powerful feature that you can also include in the `bash` startup files to add to your arsenal of custom commands.

Further Reading

- The INVOCATION section of the `bash` man page covers the `bash` startup files in gory detail.

12 – A Gentle Introduction to vi

There is an old joke about a visitor to New York City asking a passerby for directions to the city's famous classical music venue:

Visitor: Excuse me, how do I get to Carnegie Hall?

Passerby: Practice, practice, practice!

Learning the Linux command line, like becoming an accomplished pianist, is not something that we pick up in an afternoon. It takes years of practice. In this chapter, we will introduce the `vi` (pronounced "vee eye") text editor, one of the core programs in the Unix tradition. `vi` is somewhat notorious for its difficult user interface, but when we see a master sit down at the keyboard and begin to "play," we will indeed be witness to some great art. We won't become masters in this chapter, but when we are done, we will know how to play "chopsticks" in `vi`.

Why We Should Learn vi

In this modern age of graphical editors and easy-to-use text-based editors such as `nano`, why should we learn `vi`? There are three good reasons.

- `vi` is almost always available. This can be a lifesaver if we have a system with no graphical interface, such as a remote server or a local system with a broken X configuration. `nano`, while increasingly popular, is still not universal. POSIX, a standard for program compatibility on Unix systems, requires that `vi` be present.
- `vi` is lightweight and fast. For many tasks, it's easier to bring up `vi` than it is to find the graphical text editor in the menus and wait for its multiple megabytes to load. In addition, `vi` is designed for typing speed. As we will see, a skilled `vi` user never has to lift his or her fingers from the keyboard while editing.
- We don't want other Linux and Unix users to think we are cowards.

Okay, maybe two good reasons.

A Little Background

The first version of `vi` was written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems. `vi` derives its name from the word “visual,” because it was intended to allow editing on a video terminal with a moving cursor. Previous to *visual editors*, there were *line editors* that operated on a single line of text at a time. To specify a change, we tell a line editor to go to a particular line and describe what change to make, such as adding or deleting text. With the advent of video terminals (rather than printer-based terminals like teletypes), visual editing became possible. `vi` actually incorporates a powerful line editor called `ex`, and we can use line editing commands while using `vi`.

Most Linux distributions don't include real `vi`; rather, they ship with an enhanced replacement called `vim` (which is short for “vi improved”) written by Bram Moolenaar. `vim` is a substantial improvement over traditional Unix `vi` and is usually symbolically linked (or aliased) to the name `vi` on Linux systems. In the discussions that follow, we will assume that we have a program called `vi` that is really `vim`.

Starting and Stopping vi

To start `vi`, we simply enter the following:

```
[me@linuxbox ~]$ vi
```

A screen like this should appear:

```
~
~
~                         VIM - Vi Improved
~
~                          version 8.0.707
~                     by Bram Moolenaar et al.
~            Vim is open source and freely distributable
~
~                     Sponsor Vim development!
~          type  :help sponsor<Enter>    for information
~
~          type  :q<Enter>               to exit
~          type  :help<Enter>  or  <F1>  for on-line help
~          type  :help version8<Enter>   for version info
~
```

```
~                   Running in Vi compatible mode
~          type  :set nocp<Enter>        for Vim defaults
~          type  :help cp-default<Enter> for info on this
~
~
~
```

Just as we did with `nano` earlier, the first thing to learn is how to exit. To exit, we enter the following command (note that the colon character is part of the command):

```
:q
```

The shell prompt should return. If, for some reason, `vi` will not quit (usually because we made a change to a file that has not yet been saved), we can tell `vi` that we really mean it by adding an exclamation point to the command.

```
:q!
```

Tip: If you get "lost" in `vi`, try pressing the `Esc` key twice to find your way again.

Compatibility Mode

In the example startup screen above, we see the text "Running in Vi compatible mode." This means that `vim` will run in a mode that is closer to the normal behavior of `vi` rather than the enhanced behavior of `vim`. For the purposes of this chapter, we will want to run `vim` with its enhanced behavior. To do this, you have a few options. Try running `vim` instead of `vi`. If that works, consider adding `alias vi='vim'` to your `.bashrc` file. Alternatively, use this command to add a line to your `vim` configuration file:

```
echo "set nocp" >> ~/.vimrc
```

Different Linux distributions package `vim` in different ways. Some distributions install a minimal version of `vim` by default that supports only a limited set of

> `vim` features. While performing the lessons that follow, you may encounter missing features. If this is the case, install the full version of `vim`.

Editing Modes

Let's start `vi` again, this time passing to it the name of a nonexistent file. This is how we can create a new file with `vi`:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ vi foo.txt
```

If all goes well, we should get a screen like this:

```

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"foo.txt" [New File]
```

The leading tilde characters (~) indicate that no text exists on that line. This shows that we have an empty file. **Do not type anything yet!**

The second most important thing to learn about `vi` (after learning how to exit) is that `vi` is a *modal editor*. When `vi` starts, it begins in *command mode*. In this mode, almost every key is a command, so if we were to start typing, `vi` would basically go crazy and make a big mess.

Entering Insert Mode

To add some text to our file, we must first enter *insert mode*. To do this, we press the `i` key. Afterward, we should see the following at the bottom of the screen if `vim` is running in its usual enhanced mode (this will not appear in `vi` compatible mode):

```
-- INSERT --
```

Now we can enter some text. Try this:

```
The quick brown fox jumped over the lazy dog.
```

To exit insert mode and return to command mode, press the `Esc` key.

Saving Our Work

To save the change we just made to our file, we must enter an *ex command* while in command mode. This is easily done by pressing the `:` key. After doing this, a colon character should appear at the bottom of the screen.

```
:
```

To write our modified file, we follow the colon with a `w` and then press `Enter`.

```
:w
```

The file will be written to the hard drive, and we should get a confirmation message at the bottom of the screen, like this:

```
"foo.txt" [New] 1L, 46C written
```

Tip: If you read the vim documentation, you will notice that (confusingly) command mode is called *normal mode* and ex commands are called *command mode*. Beware.

Moving the Cursor Around

While in command mode, vi offers a large number of movement commands, some of which it shares with less. Table 12-1 lists a subset.

Table 12-1: Cursor Movement Keys

Key	Moves The Cursor
l or right arrow	Right one character.
h or left arrow	Left one character.
j or down arrow	Down one line.
k or up arrow	Up one line.
0 (zero)	To the beginning of the current line.
^	To the first non-whitespace character on the current line.
$	To the end of the current line.
w	To the beginning of the next word or punctuation character.
W	To the beginning of the next word, ignoring punctuation characters.
b	To the beginning of the previous word or punctuation character.
B	To the beginning of the previous word, ignoring punctuation characters.
Ctrl-f or Page Down	Down one page.
Ctrl-b or Page Up	Up one page.
*number*G	To line *number*. For example, 1G moves to the first line of the file.
G	To the last line of the file.

Why are the `h`, `j`, `k`, and `l` keys used for cursor movement? When `vi` was originally written, not all video terminals had arrow keys, and skilled typists could use regular keyboard keys to move the cursor without ever having to lift their fingers from the keyboard.

Many commands in `vi` can be prefixed with a number, as with the "G" command listed above. By prefixing a command with a number, we may specify the number of times a command is to be carried out. For example, the command "5j" causes `vi` to move the cursor down five lines.

Basic Editing

Most editing consists of a few basic operations such as inserting text, deleting text, and moving text around by cutting and pasting. `vi`, of course, supports all of these operations in its own unique way. `vi` also provides a limited form of undo. If we press the "u" key while in command mode, `vi` will undo the last change that you made. This will come in handy as we try some of the basic editing commands.

Appending Text

`vi` has several different ways of entering insert mode. We have already used the `i` command to insert text.

Let's go back to our `foo.txt` file for a moment.

```
The quick brown fox jumped over the lazy dog.
```

If we wanted to add some text to the end of this sentence, we would discover that the `i` command will not do it, since we can't move the cursor beyond the end of the line. `vi` provides a command to append text, the sensibly named `a` command. If we move the cursor to the end of the line and type `a`, the cursor will move past the end of the line and `vi` will enter insert mode. This will allow us to add some more text.

```
The quick brown fox jumped over the lazy dog. It was cool.
```

Remember to press the `Esc` key to exit insert mode.

Since we will almost always want to append text to the end of a line, `vi` offers a shortcut to move to the end of the current line and start appending. It's the `A` command. Let's try it and add some more lines to our file.

First, we'll move the cursor to the beginning of the line using the “0” (zero) command. Now we type A and add the following lines of text:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

Again, press the Esc key to exit insert mode.

As we can see, the “A” command is more useful as it moves the cursor to the end of the line before starting insert mode.

Opening a Line

Another way we can insert text is by “opening” a line. This inserts a blank line between two existing lines and enters insert mode. This has two variants as described in Table 12-2.

Table 12-2: Line Opening Keys

Command	Opens
o	The line below the current line
O	The line above the current line

We can demonstrate this as follows: place the cursor on “Line 3” then type o.

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3

Line 4
Line 5
```

A new line was opened below the third line and we entered insert mode. Exit insert mode by pressing the Esc key. Press the u key to undo our change.

Press the O key to open the line above the cursor:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2

Line 3
Line 4
Line 5
```

Exit insert mode by pressing the `Esc` key and undo our change by pressing `u`.

Deleting Text

As we might expect, `vi` offers a variety of ways to delete text, all of which contain one of two keystrokes. First, the `x` command will delete a character at the cursor location. `x` may be preceded by a number specifying how many characters are to be deleted. The `d` command is more general purpose. Like `x`, it may be preceded by a number specifying the number of times the deletion is to be performed. In addition, `d` is always followed by a movement command that controls the size of the deletion. Table 12-3 provides some examples:

Table 12-3: Text Deletion Commands

Command	Deletes
`x`	The current character
`3x`	The current character and the next two characters
`dd`	The current line
`5dd`	The current line and the next four lines
`dW`	From the current cursor position to the beginning of the next word
`d$`	From the current cursor location to the end of the current line
`d0`	From the current cursor location to the beginning of the line
`d^`	From the current cursor location to the first non-whitespace character in the line
`dG`	From the current line to the end of the file
`d20G`	From the current line to the twentieth line of the file

Place the cursor on the word `It` on the first line of our text. Press the `x` key repeatedly until the rest of the sentence is deleted. Next, press the `u` key repeatedly until the deletion is undone.

> **Note:** Real `vi` supports only a single level of undo. `vim` supports multiple levels.

Let's try the deletion again, this time using the `d` command. Again, move the cursor to the word `It` and type `dW` to delete the word.

```
The quick brown fox jumped over the lazy dog. was cool.
Line 2
Line 3
Line 4
Line 5
```

Type `d$` to delete from the cursor position to the end of the line.

```
The quick brown fox jumped over the lazy dog.
Line 2
Line 3
Line 4
Line 5
```

Press `dG` to delete from the current line to the end of the file.

```
~
~
~
~
~
```

Press `u` three times to undo the deletion.

Cutting, Copying, and Pasting Text

The `d` command not only deletes text, it also “cuts” text. Each time we use the `d` command, the deletion is copied into a paste buffer (think clipboard) that we can later recall with the `p` command to paste the contents of the buffer after the cursor or with the `P` command to paste the contents before the cursor.

The `y` command is used to “yank” (copy) text in much the same way the `d` command is used to cut text. Table 12-4 provides some examples of combining the `y` command with various movement commands:

Table 12- 4: Yanking Commands

Command	Copies
`yy`	The current line
`5yy`	The current line and the next four lines
`yW`	From the current cursor position to the beginning of the next word
`y$`	From the current cursor location to the end of the current line
`y0`	From the current cursor location to the beginning of the line
`y^`	From the current cursor location to the first non-whitespace character in the line
`yG`	From the current line to the end of the file
`y20G`	From the current line to the twentieth line of the file

Let's try some copy-and-paste. Place the cursor on the first line of the text and type `yy` to copy the current line. Next, move the cursor to the last line (`G`) and type `p` to paste the line below the current line.

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog. It was cool.
```

Just as before, the `u` command will undo our change. With the cursor still positioned on the last line of the file, type `P` to paste the text above the current line.

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
The quick brown fox jumped over the lazy dog. It was cool.
Line 5
```

Try some of the other `y` commands in the Table 12-4 and get to know the behavior of both the `p` and `P` commands. When you are done, return the file to its original state.

Joining Lines

`vi` is rather strict about its idea of a line. Normally, it is not possible to move the cursor to the end of a line and delete the end-of-line character to join one line with the one below it. Because of this, `vi` provides a specific command, `J` (not to be confused with `j`, which is for cursor movement), to join lines together.

If we place the cursor on `Line 3` and type the `J` command, here's what happens:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3 Line 4
Line 5
```

Search-and-Replace

`vi` has the ability to move the cursor to locations based on searches. It can do this either on a single line or over an entire file. It can also perform text replacements with or without confirmation from the user.

Searching Within a Line

The `f` command searches a line and moves the cursor to the next instance of a specified character. For example, the command `fa` would move the cursor to the next occurrence of the character `a` within the current line. After performing a character search within a line, the search may be repeated by typing a semicolon.

Searching the Entire File

To move the cursor to the next occurrence of a word or phrase, the `/` command is used.

This works the same way as we learned earlier in the `less` program. When you type the `/` command, a `/` will appear at the bottom of the screen. Next, type the word or phrase to be searched for, followed by the `Enter` key. The cursor will move to the next location containing the search string. A search may be repeated using the previous search string with the `n` command. Here's an example:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

Place the cursor on the first line of the file. Type followed by the `Enter` key.

```
/Line
```

The cursor will move to line 2. Next, type `n` and the cursor will move to line 3. Repeating the `n` command will move the cursor down the file until it runs out of matches. While we have so far used only words and phrases for our search patterns, `vi` allows the use of *regular expressions*, a powerful method of expressing complex text patterns. We will cover regular expressions fully in chapter 19, "Regular Expressions."

Global Search-and-Replace

`vi` uses an ex command to perform search-and-replace operations (called *substitution* in `vi`) over a range of lines or the entire file. To change the word `Line` to `line` for the entire file, we would enter the following command:

```
:%s/Line/line/g
```

Let's break down this command into separate items and see what each one does (see Table 12-5).

Table 12- 5:An Example of Global Search-and-Replace Syntax

Item	Meaning
`:`	The colon character starts an ex command.
`%`	This specifies the range of lines for the operation. `%` is a

	shortcut meaning from the first line to the last line. Alternately, the range could have been specified `1,5` (since our file is five lines long) or `1,$`, which means “from line 1 to the last line in the file.” If the range of lines is omitted, the operation is performed only on the current line.
`s`	This specifies the operation. In this case, it’s substitution (search-and-replace).
`/Line/line/`	This specifies the search pattern and the replacement text.
`g`	This means “global” in the sense that the search-and-replace is performed on every instance of the search string in the line. If omitted, only the first instance of the search string on each line is replaced.

After executing our search-and-replace command, our file looks like this:

```
The quick brown fox jumped over the lazy dog. It was cool.
line 2
line 3
line 4
line 5
```

We can also specify a substitution command with user confirmation. This is done by adding a `c` to the end of the command. Here’s an example:

```
:%s/line/Line/gc
```

This command will change our file back to its previous form; however, before each substitution, `vi` stops and asks us to confirm the substitution with this message:

```
replace with Line (y/n/a/q/l/^E/^Y)?
```

Each of the characters within the parentheses is a possible choice, as described in Table 12-6.

Table 12-6: Replace Confirmation Keys

Key	**Action**

`y`	Perform the substitution.
`n`	Skip this instance of the pattern.
`a`	Perform the substitution on this and all subsequent instances of the pattern.
`q` or `Esc`	Quit substituting.
`l`	Perform this substitution and then quit. This is hort for “last.”
`Ctrl-e, Ctrl-y`	Scroll down and scroll up, respectively. This is useful for viewing the context of the proposed substitution.

If you type `y`, the substitution will be performed, `n` will cause `vi` to skip this instance and move on to the next one.

Editing Multiple Files

It's often useful to edit more than one file at a time. You might need to make changes to multiple files or you may need to copy content from one file into another. With `vi` we can open multiple files for editing by specifying them on the command line.

```
vi file1 file2 file3...
```

Let's exit our existing vi session and create a new file for editing. Type `:wq` to exit `vi`, saving our modified text. Next, we'll create an additional file in our home directory that we can play with. We'll create the file by capturing some output from the `ls` command.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Let's edit our old file and our new one with `vi`.

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

`vi` will start, and we will see the first file on the screen.

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

Switching Between Files

To switch from one file to the next, use this ex command:

```
:bn
```

To move back to the previous file use the following:

```
:bp
```

While we can move from one file to another, `vi` enforces a policy that prevents us from switching files if the current file has unsaved changes. To force `vi` to switch files and abandon your changes, add an exclamation point (!) to the command.

In addition to the switching method described above, `vim` (and some versions of `vi`) provides some ex commands that make multiple files easier to manage. We can view a list of files being edited with the `:buffers` command. Doing so will display a list of the files at the bottom of the display.

```
:buffers
  1 %a   "foo.txt"                        line 1
  2      "ls-output.txt"                  line 0
Press ENTER or type command to continue
```

To switch to another buffer (file), type `:buffer` followed by the number of the buffer we want to edit. For example, to switch from buffer 1 containing the file `foo.txt` to buffer 2 containing the file `ls-output.txt` we would type this:

```
:buffer 2
```

Our screen now displays the second file. Another way we can change buffers is to use the

`:bn` (short for buffer next) and `:bp` (short for buffer previous) commands mentioned earlier.

Opening Additional Files for Editing

It's also possible to add files to our current editing session. The ex command `:e` (short for "edit") followed by a filename will open an additional file. Let's end our current editing session and return to the command line.

Start `vi` again with just one file.

```
[me@linuxbox ~]$ vi foo.txt
```

To add our second file, enter the following:

```
:e ls-output.txt
```

It should appear on the screen. The first file is still present as we can verify.

```
:buffers
  1 #    "foo.txt"                        line 1
  2 %a   "ls-output.txt"                  line 0
Press ENTER or type command to continue
```

Copying Content from One File into Another

Often while editing multiple files, we will want to copy a portion of one file into another file that we are editing. This is easily done using the usual yank and paste commands we used earlier. We can demonstrate as follows. First, using our two files, switch to buffer 1 (`foo.txt`) by entering this:

```
:buffer 1
```

That should give us this:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

Next, move the cursor to the first line, and type `yy` to yank (copy) the line.

Switch to the second buffer by entering the following:

```
:buffer 2
```

The screen will now contain some file listings like this (only a portion is shown here):

```
total 343700
-rwxr-xr-x 1 root root       31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root       25368 2016-10-06 20:16 a52dec
-rwxr-xr-x 1 root root       11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root        7292 2017-05-04 17:43 aainfo
```

Move the cursor to the first line and paste the line we copied from the preceding file by typing the `p` command.

```
total 343700
The quick brown fox jumped over the lazy dog. It was cool.
-rwxr-xr-x 1 root root       31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root       25368 2016-10-06 20:16 a52dec
-rwxr-xr-x 1 root root       11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root        7292 2017-05-04 17:43 aainfo
```

Inserting an Entire File into Another

It's also possible to insert an entire file into one that we are editing. To see this in action, let's end our `vi` session and start a new one with just a single file.

```
[me@linuxbox ~]$ vi ls-output.txt
```

We will see our file listing again.

```
total 343700
-rwxr-xr-x 1 root root        31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root         8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root       111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root        25368 2016-10-06 20:16 a52dec
-rwxr-xr-x 1 root root        11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root         7292 2017-05-04 17:43 aainfo
```

Move the cursor to the third line, and then enter the following ex command:

```
:r foo.txt
```

The `:r` command (short for "read") inserts the specified file below the cursor position. Our screen should now look like this:

```
total 343700
-rwxr-xr-x 1 root root        31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root         8240 2017-12-09 13:39 411toppm
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
-rwxr-xr-x 1 root root       111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root        25368 2016-10-06 20:16 a52dec
-rwxr-xr-x 1 root root        11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root         7292 2017-05-04 17:43 aainfo
```

Saving Our Work

Like everything else in `vi`, there are several different ways to save our edited files. We have already covered the ex command `:w`, but there are some others we may also find helpful.

In command mode, typing `ZZ` will save the current file and exit `vi`. Likewise, the ex command `:wq` will combine the `:w` and `:q` commands into one that will both save the file and exit.

The `:w` command may also specify an optional filename. This acts like "Save As..." For example, if we were editing `foo.txt` and wanted to save an alternate version called `foo1.txt`, we would enter the following:

```
:w foo1.txt
```

Note: While this command saves the file under a new name, it does not change the name of the file we are editing. As we continue to edit, we will still be editing `foo.txt`, not `foo1.txt`.

Summing Up

With this basic set of skills, we can now perform most of the text editing needed to maintain a typical Linux system. Learning to use `vim` on a regular basis will pay off in the long run. Since vi-style editors are so deeply embedded in Unix culture, we will see many other programs that have been influenced by its design. `less` is a good example of this influence.

Further Reading

Even with all that we have covered in this chapter, we have barely scratched the surface of what `vi` and `vim` can do. Here are a couple of on-line resources you can use to continue your journey towards `vi` mastery:

- *Vim, with Vigor* – A follow up tutorial to this one on LinuxCommand.org that brings the reader up to an intermediate level of skill. You can find it at: http://linuxcommand.org/lc3_adv_vimvigor.php
- *Learning The vi Editor* – A Wikibook from Wikipedia that offers a concise guide to `vi` and several of its work-a-likes including `vim`. It's available at: http://en.wikibooks.org/wiki/Vi
- *The Vim Book* - The `vim` project has a 570-page book that covers (almost) all of the features in `vim`. You can find it at: ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf.
- A Wikipedia article on Bill Joy, the creator of `vi`.: http://en.wikipedia.org/wiki/Bill_Joy

- A Wikipedia article on Bram Moolenaar, the author of `vim`:
  http://en.wikipedia.org/wiki/Bram_Moolenaar

13 – Customizing the Prompt

In this chapter, we will look at a seemingly trivial detail—our shell prompt. This examination will reveal some of the inner workings of the shell and the terminal emulator program.

Like so many things in Linux, the shell prompt is highly configurable, and while we have pretty much taken it for granted, the prompt is a really useful device once we learn how to control it.

Anatomy of a Prompt

Our default prompt looks something like this:

```
[me@linuxbox ~]$
```

Notice that it contains our username, our hostname, and our current working directory, but how did it get that way? Very simply, it turns out. The prompt is defined by an environment variable named `PS1` (short for "prompt string 1"). We can view the contents of `PS1` with the `echo` command.

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

> **Note:** Don't worry if your results are not the same as the example above. Every Linux distribution defines the prompt string a little differently, some quite exotically.

From the results, we can see that `PS1` contains a few of the characters we see in our prompt such as the brackets, the at-sign, and the dollar sign, but the rest are a mystery. The astute among us will recognize these as *backslash-escaped special characters* like those we saw in Chapter 7, "Seeing the World as the Shell Sees It." Table 13-1 provides a partial list of the characters that the `bash` treats specially in the prompt string.

Table 13-1: Escape Codes Used in Shell Prompts

Sequence	Value Displayed
`\a`	ASCII bell. This makes the computer beep when it is encountered.
`\d`	Current date in day, month, date format. For example, “Mon May 26.”
`\h`	Hostname of the local machine minus the trailing domain name.
`\H`	Full hostname.
`\j`	Number of jobs running in the current shell session.
`\l`	Name of the current terminal device.
`\n`	A newline character.
`\r`	A carriage return.
`\s`	Name of the shell program.
`\t`	Current time in 24-hour hours:minutes:seconds format.
`\T`	Current time in 12-hour format.
`\@`	Current time in 12-hour AM/PM format.
`\A`	Current time in 24-hour hours:minutes format.
`\u`	Username of the current user.
`\v`	Version number of the shell.
`\V`	Version and release numbers of the shell.
`\w`	Name of the current working directory.
`\W`	Last part of the current working directory name.
`\!`	History number of the current command.
`\#`	Number of commands entered during this shell session.
`\$`	This displays a “$” character unless we have superuser privileges. In that case, it displays a “#” instead.
`\[`	Signals the start of a series of one or more non-printing characters. This is used to embed non-printing control characters that manipulate the terminal emulator in some way, such as moving the cursor or changing text colors.

`\]`	Signals the end of a non-printing character sequence.

Trying Some Alternative Prompt Designs

With this list of special characters, we can change the prompt to see the effect. First, we'll back up the existing prompt string so we can restore it later. To do this, we will copy the existing string into another shell variable that we create ourselves.

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

We create a new variable called `ps1_old` and assign the value of `PS1` to it. We can verify that the string has been copied by using the `echo` command.

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

We can restore the original prompt at any time during our terminal session by simply reversing the process.

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Now that we are ready to proceed, let's see what happens if we have an empty prompt string.

```
[me@linuxbox ~]$ PS1=
```

If we assign nothing to the prompt string, we get nothing. No prompt string at all! The prompt is still there, but displays nothing, just as we asked it to do. Since this is kind of disconcerting to look at, we'll replace it with a minimal prompt.

```
PS1="\$ "
```

That's better. At least now we can see what we are doing. Notice the trailing space within the double quotes. This provides the space between the dollar sign and the cursor when the prompt is displayed.

Let's add a bell to our prompt.

```
$ PS1="\[\a\]\$ "
```

Now we should hear a beep each time the prompt is displayed, though some systems disable this "feature." This could get annoying, but it might be useful if we needed notification when an especially long-running command has been executed. Note that we included the `\[` and `\]` sequences. Since the ASCII bell (`\a`) does not "print," that is, it does not move the cursor, we need to tell `bash` so it can correctly determine the length of the prompt.

Next, let's try to make an informative prompt with some hostname and time-of-day information.

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

Adding time-of-day to our prompt will be useful if we need to keep track of when we perform certain tasks. Finally, we'll make a new prompt that is similar to our original.

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Try the other sequences listed in the table above and see whether you can come up with a brilliant new prompt.

Adding Color

Most terminal emulator programs respond to certain non-printing character sequences to control such things as character attributes (such as color, bold text, and the dreaded blinking text) and cursor position. We'll cover cursor position in a little bit, but first we'll look at color.

Terminal Confusion

Back in ancient times, when terminals were hooked to remote computers, there were many competing brands of terminals and they all worked differently. They had different keyboards, and they all had different ways of interpreting control information. Unix and Unix-like systems have two rather complex subsystems to

> deal with the babel of terminal control (called `termcap` and `terminfo`). If you look in the deepest recesses of your terminal emulator settings, you may find a setting for the type of terminal emulation.
>
> In an effort to make terminals speak some sort of common language, the American National Standards Institute (ANSI) developed a standard set of character sequences to control video terminals. Old-time DOS users will remember the `ANSI.SYS` file that was used to enable interpretation of these codes.

Character color is controlled by sending the terminal emulator an *ANSI escape code* embedded in the stream of characters to be displayed. The control code does not "print out" on the display; rather, it is interpreted by the terminal as an instruction. As we saw in the table above, the `\[` and `\]` sequences are used to encapsulate non-printing characters. An ANSI escape code begins with an octal 033 (the code generated by the `Esc` key), followed by an optional character attribute, followed by an instruction. For example, the code to set the text color to normal (attribute = 0), black text is as follows:

```
\033[0;30m
```

Table 13-2 lists the available text colors. Notice that the colors are divided into two groups, differentiated by the application of the bold character attribute (1), which creates the appearance of "light" colors.

Table 13- 2: Escape Sequences Used to Set Text Colors

Sequence	Text Color	Sequence	Text Color
`\033[0;30m`	Black	`\033[1;30m`	Dark gray
`\033[0;31m`	Red	`\033[1;31m`	Light red
`\033[0;32m`	Green	`\033[1;32m`	Light green
`\033[0;33m`	Brown	`\033[1;33m`	Yellow
`\033[0;34m`	Blue	`\033[1;34m`	Light blue
`\033[0;35m`	Purple	`\033[1;35m`	Light purple
`\033[0;36m`	Cyan	`\033[1;36m`	Light cyan
`\033[0;37m`	Light gray	`\033[1;37m`	White

Let's try to make a red prompt. We'll insert the escape code at the beginning.

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

That works, but notice that all the text that we type after the prompt will also display in red. To fix this, we will add another escape code to the end of the prompt that tells the terminal emulator to return to the previous color.

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

That's better!

It's also possible to set the text background color using the codes listed Table 13-3. The background colors do not support the bold attribute.

Table 13-3: Escape Sequences Used to Set Background Color

Sequence	Background Color	Sequence	Background Color
\033[0;40m	Black	\033[0;44m	Blue
\033[0;41m	Red	\033[0;45m	Purple
\033[0;42m	Green	\033[0;46m	Cyan
\033[0;43m	Brown	\033[0;47m	Light gray

We can create a prompt with a red background by applying a simple change to the first escape code.

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

Try the color codes and see what you can create!

Note: Besides the normal (0) and bold (1) character attributes, text may be given underscore (4), blinking (5), and inverse (7) attributes. In the interests of good taste, many terminal emulators refuse to honor the blinking attribute, however.

Moving the Cursor

Escape codes can be used to position the cursor. This is commonly used to provide a clock or some other kind of information at a different location on the screen, such as in an upper corner each time the prompt is drawn. Table 13-4 lists the escape codes that position the cursor.

Table 13-4: Cursor Movement Escape Sequences

Escape Code	Action
`\033[`*l*`;`*c*`H`	Move the cursor to line *l* and column *c*
`\033[`*n*`A`	Move the cursor up *n* lines
`\033[`*n*`B`	Move the cursor down *n* lines
`\033[`*n*`C`	Move the cursor forward *n* characters
`\033[`*n*`D`	Move the cursor backward *n* characters
`\033[2J`	Clear the screen and move the cursor to the upper-left corner (line 0, column 0)
`\033[K`	Clear from the cursor position to the end of the current line
`\033[s`	Store the current cursor position
`\033[u`	Recall the stored cursor position

Using the codes in Table 13-4, we'll construct a prompt that draws a red bar at the top of the screen containing a clock (rendered in yellow text) each time the prompt is displayed. The code for the prompt is this formidable-looking string:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "
```

Table 13-5 outlines what each part of the string does.

Table 13-5: Breakdown of Complex Prompt String

Sequence	Action
`\[`	Begin a non-printing character sequence. The purpose of this is to allow `bash` to properly calculate the size of the visible prompt. Without an accurate calculation, command line editing features cannot position the cursor correctly.

`\033[s`	Store the cursor position. This is needed to return to the prompt location after the bar and clock have been drawn at the top of the screen. *Be aware that some terminal emulators do not recognize this code.*
`\033[0;0H`	Move the cursor to the upper-left corner, which is line 0, column 0.
`\033[0;41m`	Set the background color to red.
`\033[K`	Clear from the current cursor location (the top-left corner) to the end of the line. Since the background color is now red, the line is cleared to that color, creating our bar. Note that clearing to the end of the line does not change the cursor position, which remains in the upper-left corner.
`\033[1;33m`	Set the text color to yellow.
`\t`	Display the current time. While this is a "printing" element, we still include it in the non-printing portion of the prompt since we don't want `bash` to include the clock when calculating the true size of the displayed prompt.
`\033[0m`	Turn off color. This affects both the text and the background.
`\033[u`	Restore the cursor position saved earlier.
`\]`	End the non-printing characters sequence.
`<\u@\h \W>\$`	Prompt string.

Saving the Prompt

Obviously, we don't want to be typing that monster all the time, so we'll want to store our prompt someplace. We can make the prompt permanent by adding it to our `.bashrc` file. To do so, add these two lines to the file:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
<\u@\h \W>\$ "

export PS1
```

Summing Up

Believe it or not, there is much more that can be done with prompts involving shell functions and scripts that we haven't covered here, but this is a good start. Not everyone will care enough to change the prompt, since the default prompt is usually satisfactory. But for those of us who like to tinker, the shell provides the opportunity for many hours of casual fun.

Further Reading

- The *Bash Prompt HOWTO* from the Linux Documentation Project provides a pretty complete discussion of what the shell prompt can be made to do. It is available at:
  http://tldp.org/HOWTO/Bash-Prompt-HOWTO/
- Wikipedia has a good article on the ANSI Escape Codes:
  http://en.wikipedia.org/wiki/ANSI_escape_code